A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0

William G. Wood

February 2007

TECHNICAL REPORT
CMU/SEI-2007-TR-005
ESC-TR-2007-005

Software Architecture Technology Initiative

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

This report describes an example application of the Attribute-Driven Design (ADD) method developed by the Carnegie Mellon® Software Engineering Institute. The ADD method is an approach to defining a software architecture in which the design process is based on the quality attribute requirements the software must fulfill. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

The example in this report shows a practical application of the ADD method to a client-server system. In particular, this example focuses on selecting patterns to satisfy typical availability requirements for fault tolerance. The design concerns and patterns presented in this report—as well as the models used to determine whether the architecture satisfies the architectural drivers—can be applied in general to include fault tolerance in a system. Most of the reasoning used throughout the design process is pragmatic and models how an experienced architect works.

1 Introduction

This report describes the practical application of the Attribute-Driven Design (ADD) method developed by the Carnegie Mellon® Software Engineering Institute (SEI). The ADD method is an approach to defining a software architecture in which the design process is based on the quality attribute requirements the software must fulfill. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

The example in this report applies ADD to a client-server system to satisfy several architectural drivers, such as functional requirements, design constraints, and quality attribute requirements. In particular, this example focuses on selecting patterns to satisfy typical availability requirements for fault tolerance. The design concerns and patterns presented in this report—as well as the models used to determine whether the architecture satisfies the architectural drivers—can be applied in general to include fault tolerance in a system. Most of the reasoning used throughout the design process is pragmatic and models how an experienced architect works.

1.1 SUMMARY OF ADD

This example follows the most current version of the ADD method as described in the companion technical report, *Attribute-Driven Design (ADD) Version 2.0* [Wojcik 2006].[1] The eight steps of the ADD method are shown in Figure 1 on page 2.

Each step in the method is described in Section 4 of this report, "ADD Second Iteration." However, the method can be summarized as follows:

- Step 1 verifies that there are sufficient requirements. It requires that the architectural drivers (functional requirements, design constraints, and scenarios) are prioritized by the stakeholders before proceeding. However, in the example we're using, the details of the first iteration are missing. The prioritization of the architectural drivers is shown in Step 3 of the second iteration.
- Steps 2 through 7 are completed sequentially for each iteration. First, we choose an element to decompose (Step 2), and then we step through the process to complete the design. These steps are shown individually in Section 4 of this report.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[1] The book *Software Architecture in Practice* first detailed the ADD method [Bass 2003, p. 155–166].

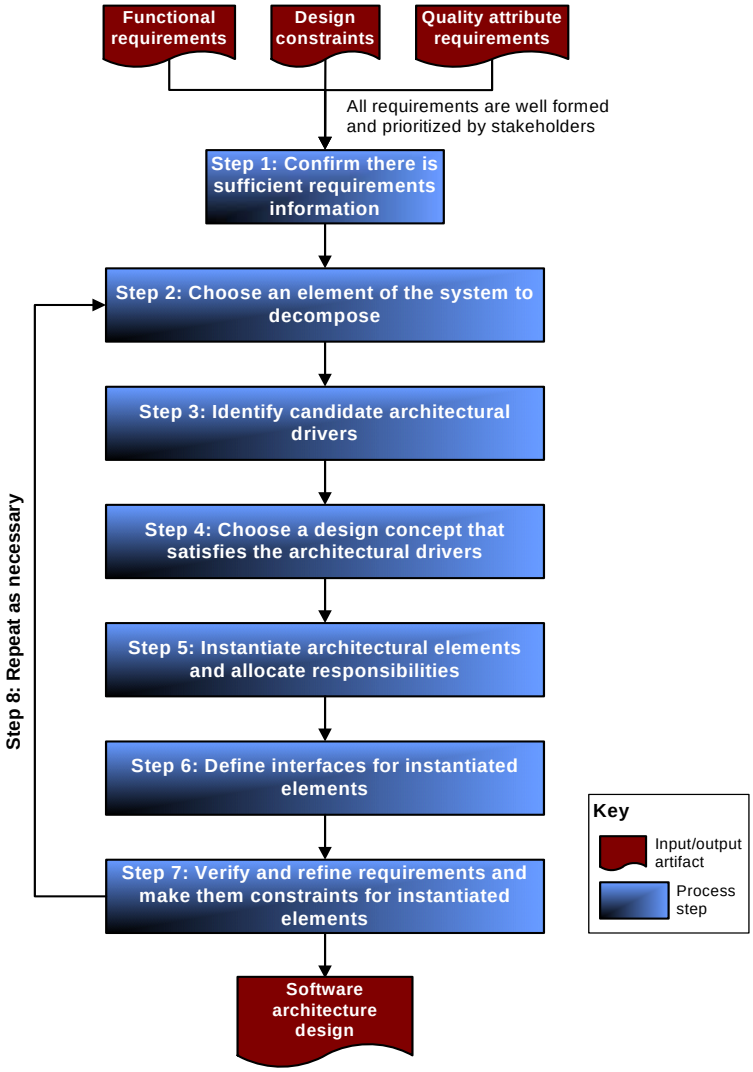

Figure 1: Steps of ADD[2]

1.2 EXAMPLE PROCESS

In the design of large-scale systems, many design activities occur in parallel with each parallel activity developing a portion of the architecture. These separate design pieces must be merged (and possibly changed) to create a consistent and complete architecture that meets all the stakeholder requirements.

In our example, we assume that the architecture team has already conducted the first iteration of ADD and developed an overview of the architecture. The results of that first iteration include placeholders for "fault-tolerance services" and other services such as "start-up services" and "persistent storage services." In Section 2 of this report, we sketch out the first iteration results to help explain the resulting architecture.

[2] This figure is copied from Wojcik's work [Wojcik 2006].

For the second iteration, the architecture team assigns the “fault-tolerance services” design requirement to a fault-tolerance expert for further refinement. This iteration, which focuses on the “fault-tolerance services” system element, is discussed in detail in Section 4. In that section, we also review the different fault-tolerance concerns, alternative design patterns, and the reasoning used to select among the alternatives to satisfy the architectural drivers. In this selection process, we must satisfy an end-to-end timing deadline starting with the failure and ending with a fully recovered system. To this end, we build a timing model to use as an analysis tool. Recall that we are drafting an architecture that satisfies the requirement that an end-to-end timing scenario be met when a fault occurs; however, we are not trying to build a complete architecture at this stage. The results of this iteration will be merged with the designs of other parallel activities.

The fault-tolerance expert is given some leeway in not satisfying all the architectural drivers. The expert may return to the architecture team and request relief from specific architectural drivers, if they force the solution space to use complex patterns that are difficult to implement. The architecture team has the opportunity to revisit some of the requirements given to the fault-tolerance expert, make changes, and allow the expert to find a more reasonable solution.

When fault tolerance is being addressed, existing services may be used in some cases, such as a health monitoring or warm restart service using a proxy. These on-hand services can become design constraints and cause the architect to make fewer design choices. In our example, however, we include the full range of design choices.

Overall, our approach to introducing fault tolerance is general and may be used as a template for designing fault-tolerance architectures.

2 System Definition

This section describes the basic client-server system in our example. We are designing its architecture in terms of three architecture requirements: functional requirements, design constraints, and quality attribute requirements.

2.1 FUNCTIONAL REQUIREMENTS

Figure 2 depicts a functional overview of our client-server example.

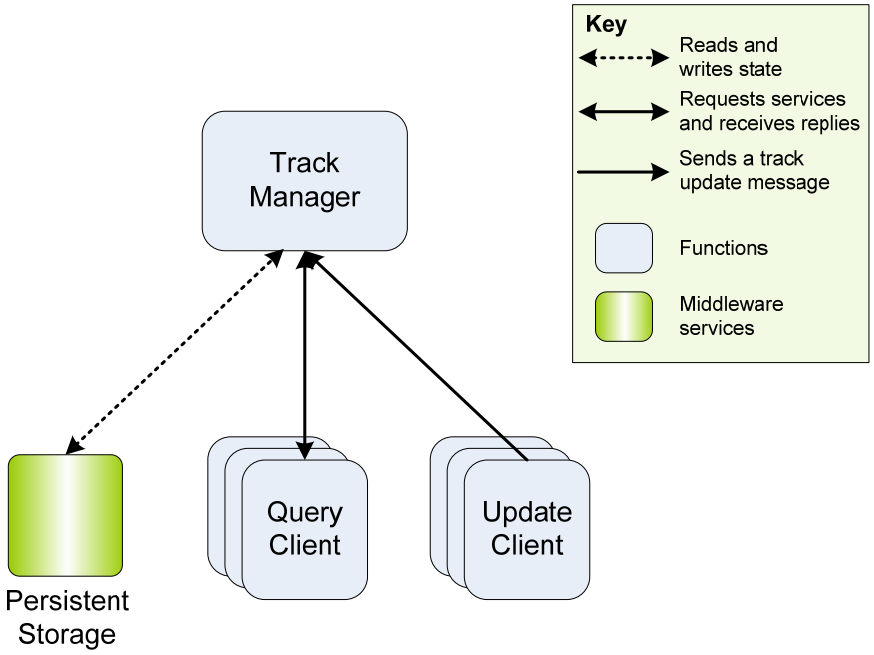

Figure 2: Functional Overview

The **Track Manager** provides a tracking service for two types of clients:

- **update clients**: These clients send track updates to the Track Manager periodically. The Track Manager can tolerate some occasional loss of updates, especially during transient conditions caused by equipment failure. All update clients perform an update every second, and the Track Manager can recover from two missed update signals when it receives the third signal. If more than two signals are missed, the operator may have to assist the Track Manager in the recovery process. In other words, if a failure occurs, the processing must restart before two seconds have elapsed in order to avoid operator intervention.
- **query clients**: These clients operate sporadically and must receive exactly one reply to their query. Query clients can be dissimilar with some clients requesting small chunks of data often (e.g., several kilobytes with five seconds between queries from a single client) and others requesting large chunks of data occasionally (e.g., several megabytes with minutes between

queries). The response time for queries should be less than double the normal response time for a particular query.

2.2 DESIGN CONSTRAINTS

Three design constraints are required:

1. **capacity restrictions:** The provided processors shall have 50% spare processor and memory capacity on delivery, and the local area network (LAN) has 50% spare throughput capability. There are 100 update clients and 25 query clients. For the purposes of timing estimates, assume that there are 100 updates and 5 queries per second.
2. **persistent storage service:** This service will maintain a copy of state that is checked at least once per minute by the Track Manager. If all replicas of the Track Manager fail, a restart can begin from the checkpoint file.
3. **two replicas:** To satisfy the availability and reliability requirements, a Reliability, Availability, and Maintainability (RMA) study has been conducted, and the Track Manager and persistent storage elements shall all have two replicas operating during normal circumstances.

2.3 QUALITY ATTRIBUTE REQUIREMENTS

The system stakeholders agree on three quality attribute scenarios that describe the various system responses to failures. These scenarios are described in Tables 1–3.

Table 1: Quality Attribute Scenario 1: Quick Recovery

Element	Statement
Stimulus	A Track Manager software or hardware component fails.
Stimulus source	A fault occurs in a Track Manager software or hardware component.
Environment	Many software clients are using this service. At the time of failure, the component may be servicing a number of clients concurrently with other queued requests.
Artifact	Track Manager
Response	All query requests made by clients before and during the failure must be honored. Update service requests can be ignored for up to two seconds without noticeable loss of accuracy.
Response measure	The secondary replica must be promoted to primary and start processing update requests within two seconds of the occurrence of a fault. Any query responses that are underway (or made near the failure time) must be responded to within three seconds of additional time (on average).

Table 2: Quality Attribute Scenario 2: Slow Recovery

Element	Statement
Stimulus	A Track Manager hardware or software component fails when no backup service is available.
Stimulus source	An error occurs in a Track Manager software or hardware component.
Environment	A single copy of the Track Manager is providing services and it fails. A spare processor is available that does not contain a copy of this component. A copy of the component is available on persistent storage and can be transferred to the spare processor via the LAN.
Artifact	Track Manager
Response	The clients are informed that the service has become unavailable. A new copy of the service is started and becomes operational. The state of the component on restart may differ from that of the failed component but by no more than one minute. The clients are informed that the service is available to receive update signals. For some tracks, the new updates can be automatically correlated to the old tracks. For others, an administrator assists in this correlation. New tracks are started when necessary. The clients are then informed that the service is available for new queries.
Response measure	The new copy is available within three minutes.

Table 3: Quality Attribute Scenario 3: Restart

Element	Statement
Stimulus	A new replica is started as the standby.
Stimulus source	The system resource manager starts the standby.
Environment	A single replica is servicing requests for service under normal conditions. No other replica is present.
Artifact	New replica of the Track Manager
Response	The initialization of the new replica has a transient impact on service requests that lasts for less than two seconds.
Response measure	The initialization of the new replica has a transient impact on service requests that lasts for less than two seconds.

3 Applying ADD

It takes at least two iterations through the ADD process to develop an architecture that satisfies the architectural requirements of a proposed system. As shown in Figure 1 on page 2, Step 1 (described below) is conducted only once to ensure that the information you have about the requirements is sufficient. We do not discuss the design steps in the first iteration, since our primary interest is in fault tolerance. The architecture team created the architectural views shown in Figure 3 and outlined in Section 3.2. A "fault-tolerance services" element is included in this view. This element is assigned to a fault-tolerance expert for design in parallel with some other designs (for example, start-up services), which we do not describe here. The fault-tolerance expert proceeds with the second iteration of the ADD method and decomposes different aspects of the fault-tolerance services.

3.1 STEP 1 OF THE ADD METHOD: CONFIRM THERE IS SUFFICIENT REQUIREMENTS INFORMATION

Section 2 (see page 5) lists the requirements for the example, which consist of functional requirements, design constraints, and quality attribute requirements.

3.2 RESULTS OF THE FIRST ADD ITERATION

The architecture team conducts the first iteration. This iteration uses a set of architectural drivers consisting of the highest priority requirements, scenarios, and their associated design constraints. These architectural drivers, the details of the team's reasoning, and the requirements gleaned during Step 1 are not included here. The resulting architecture is shown in Figure 3 and further described in the rest of this section. In addition, the software elements are summarized in Table 6 on page 12.

1. Our design uses a client-server model where the Track Manager provides services to the update and query clients. Only the primary connectors are shown in the diagram; to simplify it, some secondary interfaces are not shown (for example, all clients, Track Manager elements, and most services would have connectors to the naming service).
2. The Track Manager has been broken into two elements: A and B. This decomposition allows two deployment strategies to be considered:
   - Strategy 1: Both elements (A and B) operate in a single processor, P1. A and B together consume 50% of the processor duty cycle to handle 100 updates and 30 queries. This strategy satisfies the system performance requirements.
   - Strategy 2: Element A is in processor P1, and element B is in processor P2. Together, they can handle 150 update clients and 50 query clients. This strategy exceeds the system performance requirements.

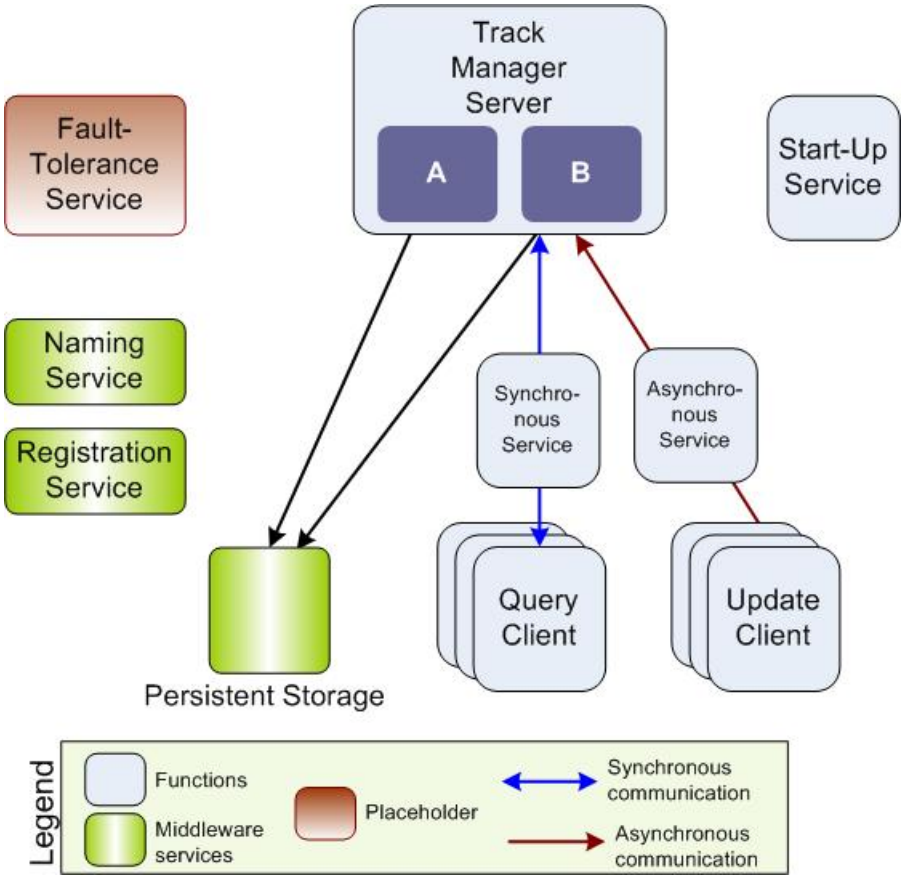

Figure 3: Software Element Primary Connectivity

The results of analyzing these design strategies are shown in Table 4. Communication system bandwidth increases by 2% when the components are placed in different processors.

Table 4: Deployment Characteristics

	P1	P2	#Updates	#Queries	P1 Load	P2 Load
Strategy 1	A, B		100	30	50%	N/A
Strategy 2	A	B	150	50	50%	30%

3. The communication mechanisms between the update and query clients and the Track Manager differ:
   - Update clients use an **asynchronous communication** mechanism. They send an update message, receive no reply, and are not suspended while the message is being delivered. (If a publish/subscribe service were available, it would be used instead.)
   - Query clients use a **synchronous communication** mechanism. They make a service request and are suspended until they receive the reply.
4. Elements A and B both contain state data that must be saved as a checkpoint in persistent storage. The elapsed times taken to copy the state to and recover the state from persistent storage are identical (see Table 5).

Table 5: Persistent Storage Elapsed Time

#	Component	Time
1	A	0.8 seconds
2	B	0.6 seconds

5. A middleware **naming service** accepts the name of a requested service and returns an access code for the service.
6. A middleware **registration service** refuses service to new clients if providing it would cause persistent storage to exceed its spare capacity limit. For simplicity's sake, the connections among clients and this service are not shown in Figure 3.
7. A separate team is assigned to consider the start-up of the Track Manager elements. The interactions between the initial designs of the start-up and fault-tolerance services will be resolved after both designs have been completed.
8. Both A and B register their interfaces with the naming service. Once again, for simplicity's sake, the connections among the functions and this service are not shown in Figure 3.
9. What happens when a service is requested from A or B for the first time depends on which type of client is making the request:
   - When an update client is making the request, the request goes directly from A or B to the asynchronous communication service and then to the naming service to get the handle for the service. (At this point, the communication mechanism caches that handle, so it doesn't have to go and get it again the next time it receives a request for that service.) Then, the communication service sends the request to A or B, appropriately. The update client can continue operation without waiting for a reply.
   - When a query client is making the request, the request goes directly from A or B to the synchronous communication service and then to the naming service to get the handle for the service. (At this point, the communication mechanism caches that handle, so it doesn't have to go and get it again the next time it receives a request for that service.) Then, the communication service sends the request to A or B, appropriately, and waits until it receives a reply. Once it does, it sends the reply to the query client. During this whole time, the query client is blocked from execution until it receives that reply.
10. The team decides to have a fault-tolerance expert refine the fault-tolerance placeholder. In fact, they suspect that the fault-tolerance placeholder is a notional concept that will permeate the system—not only will new modules have to be added, but changes to the functionality and module interfaces will also have to be made. At this point, they do not know which modules will be affected. The modules already defined, which include the placeholder (PH1) for the fault-tolerance services, are listed in Table 6.

Table 6: Elements After Iteration 1

#	Element	Fault-Tolerance Class (Yes/No)	Allocation of Architectural Drivers
1	Track Manager	Yes	N/A
2	Query clients	No	N/A
3	Update clients	No	N/A
4	Persistent storage	Yes	N/A
5	Track Manager A	Yes	Requirement 1, 3
6	Track Manager B	Yes	Requirement 1, 3
7	Synchronous communication	Yes	N/A
8	Asynchronous communication	Yes	N/A
9	Naming service	Yes	N/A
10	Registration service	Yes	N/A
PH1	Fault-tolerance service elements	Unknown	Requirement 5 Scenario 1, 2, 3 ADD iteration 1: #1, #3

The fault-tolerance expert is told to concentrate on the fault-tolerance service elements as they apply to the Track Manager elements. After this task has been completed and approved by the architecture team, the fault-tolerance considerations for the other elements, such as synchronous communications, can proceed. These elements may or may not use the same services as the Track Manager. The design of making the other elements fault tolerant is not considered here.

3.3 ORGANIZATIONAL CONSIDERATIONS

The architecture team decides to consider how to make the Track Manager fault tolerant before creating a general approach to fault tolerance. The team asks an architect with experience in fault tolerance to take this placeholder and develop a fault-tolerance architecture using these five guidelines:

- Use the requirements, the existing design of the critical element, and the scenarios as the architectural drivers for adding fault tolerance to the Track Manager.
- If, according to the fault-tolerance expert, the architectural drivers are forcing an overly complex solution, return to the architecture team with proposals to relax one or more of those drivers. The team will make the tradeoff decisions needed to achieve a simpler solution.
- Capture the rationale for the fault-tolerance architecture and the alternatives that were considered. Details about each alternative are not necessary—just the rationale used when choosing between the options.
- Don't try to address the start-up concerns. Another design team is tackling that problem. The start-up and fault-tolerance solutions will be merged at a later stage.

- **Important:** Remember that your design is preliminary and will be merged with other designs proceeding in parallel. Do not develop a complete design. Stop when you are confident that your approach will satisfy the architectural drivers; for example, do not build a complete set of sequence diagrams or other UML diagrams.

4 ADD Second Iteration

4.1 STEP 1 OF ADD: CONFIRM THERE IS SUFFICIENT REQUIREMENTS INFORMATION

This step is not necessary during each iteration. It was done once at the beginning of the ADD process.

4.2 STEP 2 OF ADD: CHOOSE AN ELEMENT OF THE SYSTEM TO DECOMPOSE

The fault-tolerance services element is chosen as the system element to decompose. Specifically, the Track Manager is targeted, since it is the system's primary element. As you can see in Table 7, other elements in the system must also be fault tolerant; however, the design team wanted to know the architectural impact of making the Track Manager fault tolerant before considering the other elements. This decision, of course, could lead to backtracking in later ADD iterations if a different scheme is needed to add fault tolerance to the other elements.

4.3 STEP 3 OF ADD: IDENTIFY CANDIDATE ARCHITECTURAL DRIVERS

Ten drivers and their priorities are listed below in Table 7. Seven drivers are identified from the initial pool of architecture requirements. Three are identified from the design constraints resulting from the first iteration of ADD.

Consider the following points as you read Table 7:

- Drivers labeled (high, high) bear directly on the end-to-end timing requirement of two seconds in scenario 1. This condition is the most difficult to satisfy and has the highest priority drivers.
- Drivers labeled (medium, medium) are associated with the timing when a single copy of the Track Manager is operating, and restoration should occur within two minutes.
- The restart scenario is least important, and a separate "start-up" design effort is considering its details. Hence, #3 drivers do not impact the design and are crossed out in the table. As a result, only nine architectural drivers should be considered.

Table 7: Architectural Driver Priorities

#	Architectural Drivers	Section Discussed In	Importance	Difficulty
1	Scenario 1 Quick Recovery	2.3	high	high
2	Scenario 2 Slow Recovery	2.3	medium	medium
~~3~~	~~Scenario 3~~ ~~Restart~~	~~2.3~~	~~low~~	~~low~~
4	Requirement 1 Track Manager Functionality	2.1	high	high
5	Design Constraint 1 Capacity Restrictions	2.2	high	high
6	Design Constraint 2 Persistent Storage Service	2.2	medium	low
7	Design Constraint 3 Two Replicas	2.2	high	high
8	ADD Step 1, #2 Deployment Characteristics	3.2	high	high
9	ADD Step 1, #3 Communication Mechanisms	3.2	high	low
10	ADD Step 1, #4 Checkpoint Timing	3.2	high	high

4.4 STEP 4 OF ADD: CHOOSE A DESIGN CONCEPT THAT SATISFIES THE ARCHITECTURAL DRIVERS

This step is the first design step in the ADD method.

Note: This section cross-references Section 7.1 of the SEI technical report titled *Attribute-Driven Design (ADD), Version 2.0* [Wojcik 2006]. This step is the heart of that document; it's where most of the design alternatives are listed, the preferred patterns are selected, an evaluation is done to validate the design, and changes are made to correct for detected deficiencies. Within Section 7.1, there are six enumerated paragraphs. To simplify your cross-referencing, each of those paragraphs is referred to in the headings of this report as ADD substep 1, 2…and so forth.

4.4.1 Step 4, Substep 1 of ADD: Identify Design Concerns

The three design concerns associated with fault-tolerance services are[3]

- **fault preparation:** This concern consists of those tactics performed routinely during normal operation to ensure that when a failure occurs, a recovery can take place.
- **fault detection:** This concern consists of the tactics associated with detecting the fault and notifying an element to deal with the fault.
- **fault recovery:** This concern addresses operations during a transient condition—the time period between the fault occurrence and the restoration of normal operation.

Table 8 shows these concerns and their breakdown into subordinate concerns, the sections in this report where the alternate patterns are listed and the selections are made.

Table 8: Design Concerns

Design Concerns	Subordinate Concerns	Alternative Patterns Section	Selecting Design Pattern Section
Fault Preparation	Restart	4.4.2.1	4.4.3.1
	Deployment	4.4.2.2	4.4.3.2
	Data integrity	4.4.2.3	4.4.3.3
Fault Detection	Health monitoring	4.4.2.4	4.4.3.4
Fault Recovery	Transparency to clients	4.4.2.5	4.4.3.5
	Start new replica	4.4.2.6	4.4.3.6
	Update client behavior after transient failure	4.4.2.7	4.4.3.7
	Update client behavior after hard failure	4.4.2.8	4.4.3.8
	Query client behavior after transient failure	4.4.2.9	4.4.3.9
	Query client behavior after hard failure	4.4.2.10	4.4.3.10

[3] We **derived** these concerns from the book titled *Software Architecture in Practice Second Edition* [Bass 2003, p. 101–105]. That book also notes a fault prevention concern, but it was not needed in our example.

4.4.2 Step 4, Substep 2 of ADD: List Alternative Patterns for Subordinate Concerns

4.4.2.1 Restart

Four design alternatives for restarting a failed component are shown below in Table 9. Two discriminating parameters are related to these patterns:

- the downtime that can be tolerated after failure (scenario 1)
- the manner in which the system treats requests for services in the time interval around the failure time; for example, if it honors them and degrades the response time or it drops them (scenario 1)

Table 9 also lists "reasonable" downtime estimates (based on experience) of these discriminating parameters.

Table 9: Restart Patterns

#	Pattern Name	Replica Type	Downtime Estimates	Loss of Services
1	Cold Restart	Passive	> 2 minutes	Yes
2	Warm Standby	Passive	> 0.3 seconds	Perhaps
3	Master/Master	Active	> 50 milliseconds	No
4	Load Sharing	Active	> 50 milliseconds	No

4.4.2.2 Deployment

The two components can be deployed with (1) both primaries on one processor and both secondaries on the second processor or (2) each primary on a different processor. The primaries are denoted by A and B; the secondaries by A' and B'. The failure condition for B mimics that of A and is not recorded in the table.

The two discriminating parameters are

- the downtime that can be tolerated after failure (scenario 1)
- the support of 100 update clients and 25 query clients (requirement 2)

Table 10: Deployment Patterns

#	Pattern Name	P #1	P #2	A Fails	# Updates	# Queries	State Recovery Time
1	Together	A, B	A', B'	A', B'	100	30	1.4
2	Apart	A, B'	A', B	A', B	150	50	0.8

4.4.2.3 Data Integrity

The data integrity tactic ensures that when a failure occurs, the secondary has sufficient state information to proceed correctly. The patterns are shown in Table 11.

Table 11: Data Integrity Patterns

#	Pattern Name	Communication Loading	Standby Processor Loading
1	Slow Checkpoint	1.2 seconds every minute	None
2	Fast Checkpoint	1.2 seconds every 2 seconds	None
3	Checkpoint + Log Changes	1.2 seconds per minute + 100 messages per second	None
4	Checkpoint + Bundled Log Changes	1.2 seconds per minute + 1 message per *x* seconds	None
5	Checkpoint + Synchronize Primary and Backup	1.2 seconds every minute + 1 message per *x* seconds	Execute to keep an updated copy of the state

4.4.2.4 Health Monitoring

A single health monitoring tactic should be considered for fault detection. Table 12 lists the patterns to consider and their discriminating parameters.

Table 12: Fault Detection Patterns

#	Pattern Name	Communication Line Loading
1	Heartbeat	4 messages (for A, A', B, B')
2	Ping/Echo	8 messages (ping and echo for A, A', B, B')
3	Update Client Detects Failure	0 messages
4	Query Client Detects Failure	0 messages

4.4.2.5 Transparency to Clients

We list three alternatives to make faults transparent to the clients in Table 13 below. Pattern 1 has no transparency, but patterns 2 and 3 provide transparency.

Table 13: Transparency Patterns

#	Pattern Name	Protocol Required	Timeout Location
1	Client Handles Failure	Unicast	Client
2	Handles Failure Proxy	Unicast	Proxy
3	Infrastructure Handles Failure	Multicast	Within the infrastructure

4.4.2.6 Start New Replica

This step is postponed, since it is closely related to the start-up mechanism that is being explored by another team.

4.4.2.7 Update Client Behavior After Transient Failure

The operation of the proxy service when a transient failure occurs has already been defined: The health monitor informs the proxy service of the failure. Then, this service sends a new secondary access code to each asynchronous communication mechanism. This access code will be used for the next update request. Essentially, this mechanism promotes the secondary to the primary.

4.4.2.8 Update Client Behavior After a Hard Failure

When a primary fails and no secondary is available, one of the design patterns in Table 14 could be used.

Table 14: Patterns for Update Client Behavior After a Hard Failure

#	Pattern Name	Impact
1	Continue to Send Updates	Unusable data is sent.
2	Stop Sending Updates	The communication line loading during downtime is saved.
3	Save Updates in a File	Larger messages are loaded on start-up.

4.4.2.9 Query Client Behavior After Transient Failure

The operation of the proxy service when such a failure occurs has already been defined: The health monitor informs the proxy service of the failure. Then, this service sends a new secondary access code to each synchronous communication mechanism. If no outstanding service request is underway, the mechanism will use this access on the next request. If a service request is underway, a new request will be issued to the new access code. It is possible for the synchronous communication to receive multiple replies (a delayed one from the primary and one from the promoted secondary). It must be able to discard the second reply.

4.4.2.10 Query Client Behavior After a Hard Failure

When a primary fails and no secondary is available, the query clients will be informed and can adjust their behavior appropriately. In that case, one of the design patterns in Table 15 could be used.

Table 15: Patterns for Query Client Behavior After a Hard Failure

#	Pattern Name	Impact
1	Continue to Send Queries	Unusable data is sent.
2	Stop Sending Queries	The communication line loading during downtime is saved.
3	Save Queries in a File	Larger messages are loaded on start-up.

4.4.3 Step 4, Substep 3 of ADD: Select Patterns from the List

This activity involves selecting a pattern from the list for each set of alternative patterns. When making your selection, you reason about which alternative is most suitable. In our example, the selections were made independently. In some cases, reasonable values were chosen as design parameters, such as heartbeat and checkpoint frequencies. In the rest of this section, we consider restart, deployment, data integrity, fault detection, transparency to client, start new replica, and client behavior after transient and hard failures. For each item, we record our reasoning, decision, and the implications of that decision.

The ADD method calls for the development of a matrix showing the interrelationships between patterns and their pros and cons on each architectural driver. It assumes that there will be a reasonable number of possible patterns and that a table is a good way to show the alternatives. Unfortunately, the inclusion of all fault tolerances as a single Step-4 design decision creates a total of 23 patterns—too many to represent in a single table. Hence, each alternative (restart, deployment, etc.) is considered separately. The pros and cons in the table are considered in separate sections below. Each section has three parts: (1) a **reasoning** paragraph describing the pros and cons for each pattern, (2) a **decision** statement emphasizing the pattern chosen, and (3) an **implication** statement showing the impact of this decision, including any obvious restrictions on choices not yet made.

4.4.3.1 Restart

Reasoning

Both scenario 1 and requirement 1 indicate that the restart time must be less than two seconds; thus, Cold Restart pattern is inappropriate (see Table 9 on page 18). The Warm Standby pattern seems to easily satisfy the timing requirement described in scenario 1. Hence it is chosen, since it is simpler to implement than the Master/Master or Load Sharing patterns.

Decision

Use the Warm Standby pattern.

Implications

1. A primary Track Manager for each component (A and B) receives all requests and responds to them.
2. A secondary (standby) Track Manager for each component (A' and B') is loaded on another processor and takes up memory.

4.4.3.2 Deployment

Reasoning

The architect is familiar with having a single failover scheme for recovery from a software or hardware failure. Hence, he chooses the first Together pattern (see Table 10 on page 18), even though it has a slower recovery time since the states for *both* A and B must be read from persistent storage, rather than just A. This pattern meets the processing requirements, although it can perform less processing. Note that the granularity of recovery differs from the granularity of failure, in that A and B must both recover when either one fails.

Decision

Use the Together pattern with both primary components that share a processor. Clearly, this option is suboptimal, since it offers reduced capability and increased recovery time. However, it was chosen for reasons of familiarity.

Implications

1. The primary components (A and B) share a processor, as do the secondary components (A' and B').
2. The system will never be operational with the primary components in different processors.

4.4.3.3 Data Integrity

Refer to Table 11, "Data Integrity Patterns," on page 19.

Reasoning

1. Clearly a checkpoint of state every minute is needed to satisfy scenario 2. However, a state that is one minute old cannot satisfy scenario 1, since one minute's worth of upgrades will be ignored if only the checkpoint is used on restart. Pattern 1 is rejected.
2. Pattern 2 would satisfy the upgrade requirements of scenarios 1 and 2; however, it places an unacceptable load on the communication system. Pattern 2 is rejected.
3. Pattern 3 would satisfy scenarios 1 and 2, but—like pattern 2—it places a significant burden on the communication system. Pattern 3 is rejected.
4. Pattern 4 satisfies scenarios 1 and 2 if x is less than two seconds. It also puts a more reasonable load on the communication system. Having a bundled upgrade periodicity of two seconds appears to be satisfactory, though a more detailed check can be made later (see Section 5). Pattern 4 is ultimately selected.
5. Pattern 5 also satisfies the scenarios but is more complex, since the secondary must execute every x seconds to update its copy of the state. Recovery would be faster, though, since it would not need to read in a checkpoint of the state. Pattern 5 is rejected due to its complexity.

Decision

Use the Checkpoint + Bundled Log Changes pattern. The log files will be used as the basis for promoting the new primary.

Implications

1. The primary replica saves the state to a persistent `CheckpointFile` every minute.
2. The primary keeps a local bundled file of all state changes for two seconds. The primary sends it as a `LogFile` every two seconds.
3. The promoted primary reads in the `CheckpointFile` after it is promoted. Then it reads the `LogFile` and updates each state change as it is read.
4. Next, the promoted secondary writes the newly updated state to persistent storage.
5. The promoted secondary can now start processing updates and queries without waiting until the persistent state update has been completed.

4.4.3.4 Fault Detection

Reasoning

An approach where the clients do not detect failure is preferable, since it implies that the application developers must understand the fault-tolerance timing requirements. In comparing the two approaches (see Table 12 on page 19), the ping/echo fault detection is more complex than the heartbeat detection and requires twice the bandwidth.

Decision

Use the Heartbeat pattern. We set the heartbeat at 0.25 seconds, which yields four communication messages per second.

Implications

1. The heartbeat must be fast enough to allow the secondary to become initialized and start processing within two seconds after a failure occurs. Initializing the two checkpoint files takes 1.2 seconds. The heartbeat adds an additional 0.25 seconds, leaving 0.55 seconds spare, which seems reasonable.
2. A health monitoring element checks for the heartbeat every 0.25 seconds. When a heartbeat is not detected, the health monitor informs all the necessary elements.
3. If a primary Track Manager component detects an internal failure, the mechanism for communicating the failure is to not issue the heartbeat.

4.4.3.5 Transparency to Client

Reasoning

It is undesirable to have the clients handle failure, since this approach requires the programmer writing the client to understand the failover mechanism. The failover could be misinterpreted easily and render it less than robust.

The infrastructure has no built-in multicast capability, and adding this feature would be expensive. You can mimic a multicast with multiple unicasts, but this approach doubles the usage of the

communication system, and is therefore undesirable. (To review the pattern options, see Table 13 on page 20.)

Decision

Use the Proxy Handles Failure pattern.

Implications

1. The proxy service registers the service methods (for example, `A.a`, `A.b`, `B.c`, `B.d`) with the name server.
2. The proxy service starts the first components, registering them under different names (`AA.a`, `AA.b`, `BB.c`, and `BB.d`) and does likewise for the secondary components (`AA'.a`, `AA'.b`, `BB'.c`, and `BB'.d`).
3. The client requests a service (`A.a`). This request causes the naming service to be invoked and to return the access code for `A.a`, designated as `access(A.a)`. Next, the client invokes `access(A.a)`.
4. The proxy service (`A.a`) determines that `AA` is the primary replica and returns `access(AA.a)` to the client as a "forward request to."
5. The client invokes `access(AA.a)` and continues to do so until `AA` fails.
6. When the health monitor detects a heartbeat failure in `AA`, it informs the proxy service.
7. The proxy informs the synchronous and asynchronous elements of the failure. These elements send their query and update requests to the newly promoted primary.

4.4.3.6 Start New Replica

This step is postponed, since, in this example, it is part of the start-up mechanism being explored by another team.

4.4.3.7 Update Client Behavior After Transient Failure

A transient failure occurs when the primary fails and a backup is scheduled to take over. In our case, the health monitor detects the failure and informs the proxy service. The proxy sends a forward-request access code to the Synchronous Communication Service (SCS). If no requests are underway, the SCS simply uses the new access code for all future requests. If a request is underway, the SCS executes a forward request with the new access code to the new Track Manager. It is possible to get two replies: one reply from the failed Track Manager component, which was inexplicably delayed beyond the failure notification, and one from the new Track Manager. If two replies are received, the second one is discarded.

4.4.3.8 Update Client Behavior After Hard Failure

Reasoning

Scenario 2 lays the foundation for this choice (see Table 14 on page 20). We are willing to accept degraded behavior and restart; therefore, pattern 3 is unnecessary and complicated. There is no point in continuing to send updates without having a Track Manager available to receive them.

Decision

We chose the Stop Sending Updates pattern, which, when there is no Track Manager, stops sending updates until a new Track Manager becomes available.

Implications

The clients must be able to do two things: (1) accept an input informing them that the Track Manager has failed and (2) stop sending updates.

4.4.3.9 Query Client Behavior After Transient Failure

We chose the same pattern as the update client (see Section 4.4.3.7) for simplicity's sake.

4.4.3.10 Query Client Behavior After Hard Failure

We chose the same pattern as the update client (see Section 4.4.3.8) for simplicity's sake.

4.4.4 Step 4, Substep 4 of ADD: Determine Relationship Between Patterns and Drivers

A summary of the selected patterns is shown below in Table 16. In the table heading

- SC# refers to the scenario number that contributes to the selection decision.
- DC# refers to the previous design # (from iteration 1) that contributed to the selection.

Table 16: Pattern/Driver Mapping

#	Pattern Types	Pattern Selected	Architectural Driver
0	# Replicas	Two Replicas	Two Replicas (DC#3)
1	Restart	Warm Standby	Two Replicas (DC#3) Quick Recovery (SC#1)
2	Deployment	Distributed	Capacity Restriction (DC#1)
3	Data Integrity	Checkpoint + Bundled Log Changes	Persistent Storage Service (DC#2) Capacity Restrictions (DC#1) Quick Recovery (SC#1) Slow Recovery (SC#2)
4	Fault Detection	Heartbeat	Capacity Restriction (DC#1) Quick Recovery (SC#1) Other- see note below
5	Transparency to Clients	Proxy Handles Failure	Capacity Restriction (DC#1) Other–*see note below*
6	New Replica	N/A	N/A
7	Update Client Behavior-Transient	Proxy Handles Failure	N/A
8	Update Client Behavior-Hard	Stop Sending Updates	Capacity Restriction (DC#1)

Table 16: Pattern/Driver Mapping (cont'd.)

#	Pattern Types	Pattern Selected	Architectural Driver
9	Query Client Behavior-Transient	Proxy Handles Failure	N/A
10	Query Client Behavior-Hard	Stop Sending Queries	Capacity Restriction (DC#1)

Note: There are numerous examples of decisions being made based on the architect's experience and preference rather than on a specific architectural driver. For example, in the fault detection selection in Section 4.4.3.4, the architect considered it inappropriate for clients to detect failure.

4.4.5 Step 4, Substep 5 of ADD: Capture Preliminary Architectural Views

In this section, we present preliminary architectural views including

- a table of system elements and the ADD iteration in which it is developed
- a functional view of the architecture
- a sequence diagram for a query client's access to data

4.4.5.1 List of the Elements

Table 17 lists the system elements and the ADD iteration in which they're developed.

Table 17: System Elements and the ADD Iteration in Which They're Developed

#	This Element	Is Developed in This ADD Iteration
1	Track Manager	Requirement
2	Query clients	Requirement
3	Update clients	Requirement
4	Persistent storage	Requirement
5	Track Manager A	1
6	Track Manager B	1
7	Synchronous communications	1
8	Asynchronous communications	1
9	Naming service	1
10	Registration service	1
11	Health monitor	2
12	Proxy server	2
13	CheckpointFileA	2

Table 17: System Elements and the ADD Iteration in Which They're Developed (cont'd.)

#	This Element	Is Developed in This ADD Iteration
14	CheckpointFileB	2
15	LogFileA	2
16	LogFileB	2

4.4.5.2 A Software Element View of the Architecture

Figure 4 shows a functional view of the software elements in the architecture and their relationships.

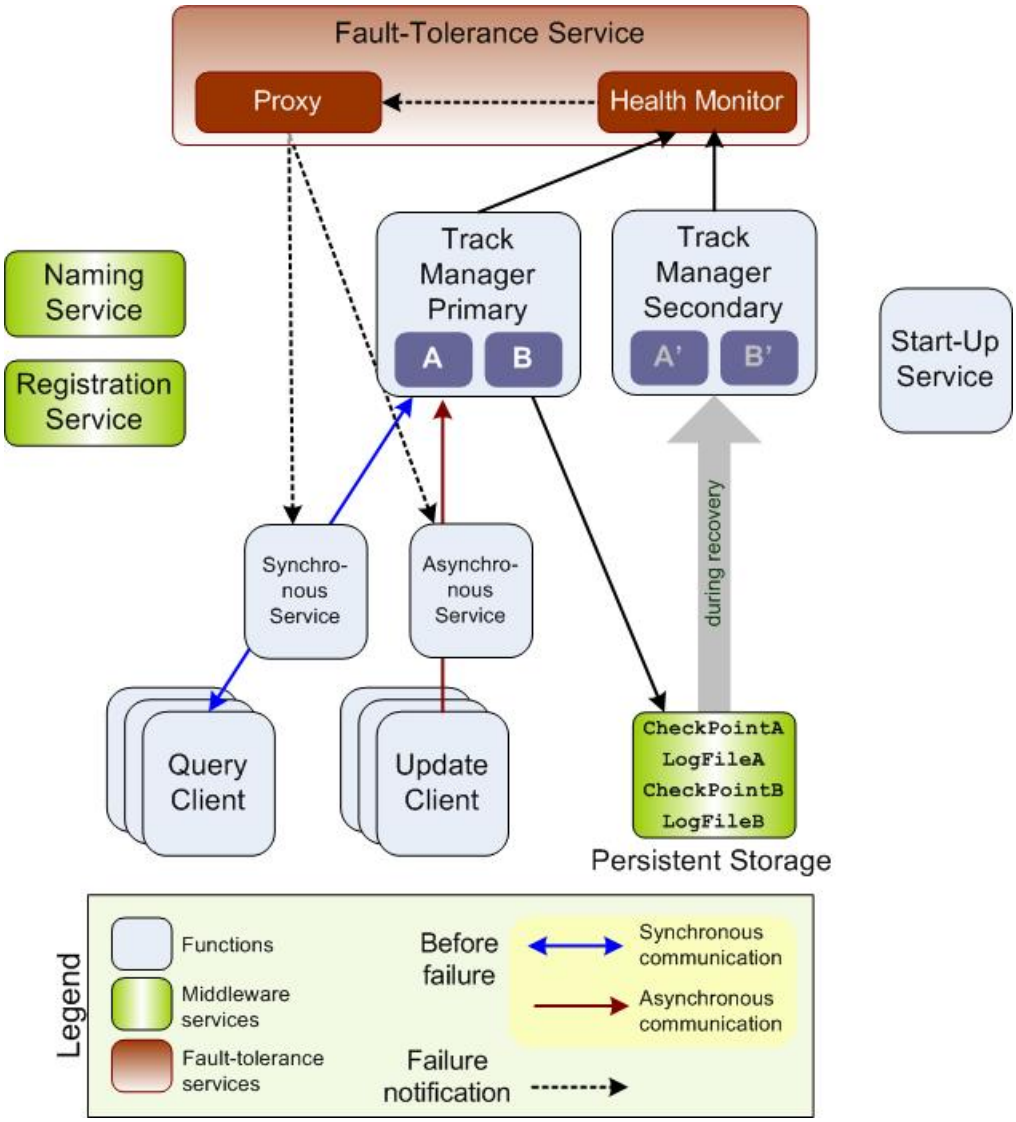

Figure 4: Software Element View of the Architecture

4.4.5.3 Sequence Diagram

The sequence diagram for a query client's access to data from client A is shown in Figure 5. The figure depicts two sequences:

1. For the first request, the synchronous communication service sends the service request to the proxy. The proxy returns a "forward request to A" message. The synchronous communication service caches the forward request to A and uses it for all future requests.
2. If A fails to issue a heartbeat to the health monitor, the latter informs the proxy that A has failed. The proxy sends a "forward request to A' " message to the synchronous communication service. The service then forwards the request to A', caches the request, and continues to send messages to A'.

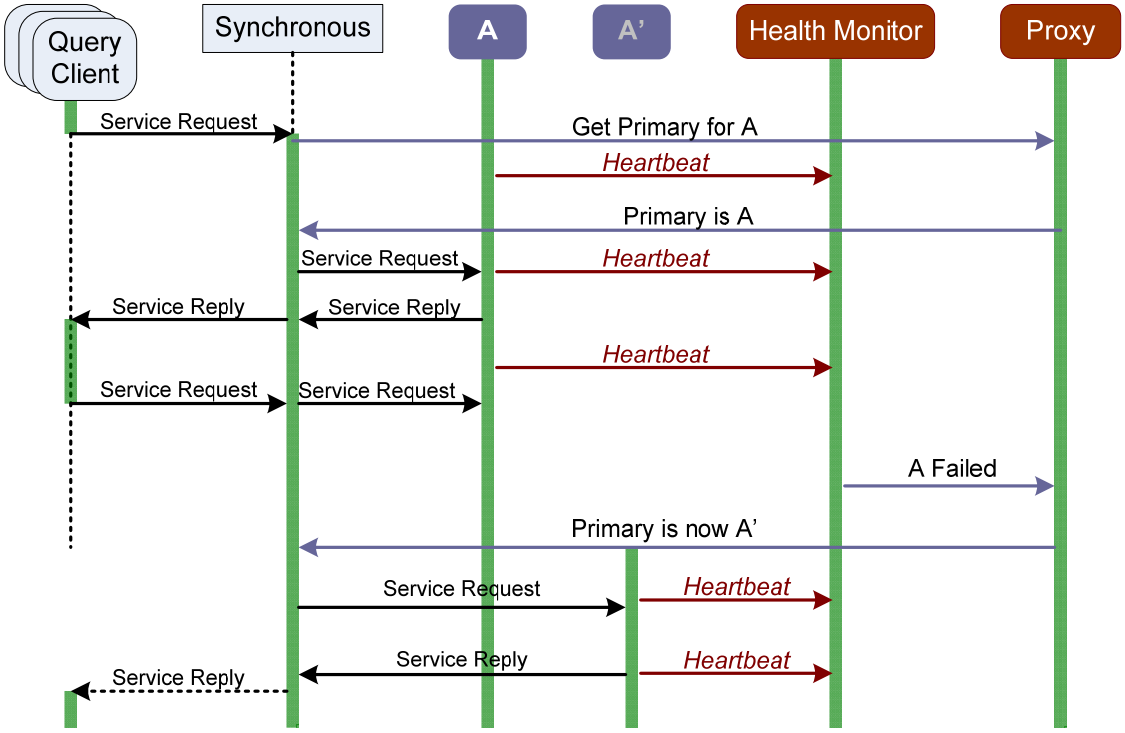

Figure 5: A Sequence Diagram of Failover from A to A'

4.4.6 Step 4.6 of ADD: Evaluate and Resolve Inconsistencies

In an architecture evaluation, the architect builds models to describe the system's behavior. The architect then analyzes these models to ensure that they satisfy the architectural drivers. In our example, we develop a timeline showing the operation around the time of failure.

Figure 6 models the operation of the system over a time period that includes a failure.

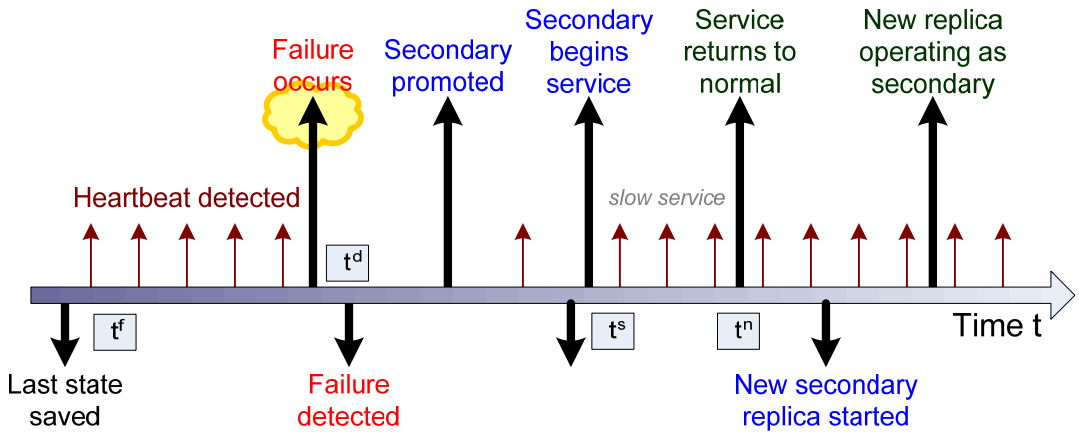

Figure 6: Timing Model

The following nine events, which occur in this order, are depicted in Figure 6.

1. A save is made of state updates to the persistent `LogFile`.
2. A heartbeat is detected a number of times after the state save.
3. A crash failure occurs in the Track Manager.
4. The health monitor detects the failure when a timeout occurs before the heartbeat.
5. The secondary Track Manager is promoted to primary.
6. The secondary service starts to respond to client requests, working off the backlog of requests and giving slower response times.
7. The service returns to normal when the transient period of slow responses ends.
8. A new replica completes initialization and is ready to synchronize with the current primary and become the secondary.
9. The new replica has completed any needed state updates, and the process of restoring the service is completed.

Six important timing aspects of the system are shown in Figure 6:

- **T_{ps}:** periodicity of the state `LogFile` save (2 seconds, see Section 4.4.3.3)
- **T_h:** periodicity of the heartbeat (0.25 seconds, see Section 4.4.3.4)
- **T_{rA}:** elapsed time taken to recover the state of A from persistent storage (0.8 seconds, see Table 5 on page 11)
- **T_{rB}:** elapsed time taken to recover the state of B from persistent storage (0.6 seconds, see Table 5 on page 11)
- **T_{rL}:** elapsed time to recover the `LogFile` from persistent storage (estimated at 0.2 seconds)
- **T_{us}:** elapsed time to update the state of A and B from the `LogFile` (estimated at 0.1 second)

The worst-case total time (T1) until the Track Manager recovery occurs when the failure is just after a heartbeat and just before the next write of the updates to the `LogFile`. In this case, the time would be

$$T1 = T_{ps} + T_h + T_{rA} + T_{rB} + T_{rL} + T_{us}$$

$$T1 = 2 + 0.25 + 0.8 + 0.6 + 0.2 + 0.1 = 3.95$$

The result is an unacceptable time of 3.95 seconds.

4.4.6.1 Resolve Timing Inconsistencies

We can improve our models in several ways, and we must make tradeoffs among these proposed improvements. Our main objective is to reduce the restart time from 3.95 seconds to less than 2 seconds, while ensuring that the communication load remains reasonable. We can modify the important timing aspects in these ways:

- Reduce the periodicity of the `LogFile` save to persistent storage. Synchronize the `LogFile` save and the heartbeat such that the heartbeat occurs just after a save is initiated; they do not need to have the same periodicity.
- Have the `LogFile` save to persistent storage serve as the heartbeat equivalent. Send the log every 0.5 seconds. Extend the persistent storage element so that it recognizes that a failure to receive the `LogFile` update triggers a request to inform the other necessary elements of a failure (i.e., proxy, standby, clients).
- Make the three persistent storage accesses concurrent instead of sequential.
- Change the deployment decision to the second pattern, in which the primaries of A and B are in different processors; hence, the failure of the processor with component A will be the worst case (since it takes more time to recover its state).
- Change the style of the state update to option 5 in Table 11 (on page 19), in which the secondary maintains a model of the state by synchronizing with the primary during start-up. It also receives a bundle of state updates periodically, thus obviating the need to read from persistent storage.
- Reduce the size of the state to be saved for components A and B by recomputing some state data on restart.

Reasoning

1. The fault-tolerance designer is reluctant to choose alternative 2, 5, or 6, since they entail requesting changes to the previous design. The expert will only do so if there is no other reasonable way to reduce the time within his or her direct control. The expert would propose such tradeoffs only after reviewing other alternatives.
2. The `LogFile` save to persistent storage can occur every second and be synchronized to occur just before every fourth heartbeat. This scheme reduces the terms ($T_{ps} + T_h$) from 2.25 seconds to 1 second. This measure is a gain of 1.25 seconds, which would reduce the response to 2.7 seconds, which is still not good enough. This measure could be further improved by reducing the periodicity to 0.5 seconds, but that option is rejected. It would cause too much additional load on the communication mechanism.

3. Access to persistent storage for all three files takes ($T_{rA} + T_{rB +} T_{rL}$) or 1.6 seconds, if done sequentially. However, if accesses are concurrent using asynchronous communications, in theory they will take only 0.8 seconds, which is the time required to get the persistent state for component A. However, a detailed analysis of the persistent storage shows that the three concurrent requests will share some resources and take 1.0 seconds. This reduction is still 0.6 seconds, which leaves us with a 2.1-second response—still not good enough.
4. The deployment decision is changed to the second option of having each component A and B in a separate processor. Hence, the worst-case access to persistent storage occurs for component A and is 0.8 seconds. If this is still done concurrently with the `Logfile` access, the total time for both will be .85 seconds. The savings in the previous step are now invalidated, and the 1.6 seconds now takes 0.85 seconds, which yields a 1.95-second response. This response does not provide quite enough of a margin.
5. The only way within the architect's control to further resolve this problem is to select alternative D and change the data integrity style. Taking this approach assumes that the primary and secondary states for A and B will not diverge in any way, which is outside of the architect's control. The architect then approaches those responsible for the previous design and explains the problem and the options. The designer team agrees to reduce the state upgrade time for component A to 0.6 seconds and the concurrent access with the `Logfile` to 0.65 seconds This change represents a further savings of 0.2 from the previous result and creates a response time of 1.75 seconds, which is within a reasonable margin.

4.4.6.2 Summary of Timing Decisions

Table 18 summarizes the timing decisions.

Table 18: Summary of Timing Decisions

#	Description	Initial Time Interval	Final Time Interval
T_{ps}	Save FileCheckPoint	2.0	1.0 (see Note 1)
T_h	Heartbeat	0.25	0.25
T_{rB}	Recover Checkpoint for B	0.6	0.0 (see Note 2)
T_{rA}	Recover Checkpoint for A	0.8	0.65 (see Note 3)
T_{rL}	Recover LogFile	0.2	0.2
T_{us}	Update state from LogFile	0.1	0.1
T	Recovery time	3.95	1.75
	Checkpoint State	60	60

Notes:

1. The heartbeat and checkpoint save are synchronized together (reasoning point 2 in Section 4.4.6.1).
2. Since A and B are in separate processors, we only have to recover the state of one of them for a single failure (reasoning point 4 in Section 4.4.6.1).
3. The state recovery and checkpoint recovery are performed concurrently (reasoning point 4 and 5 in Section 4.4.6.1).

4.5 STEP 5 OF ADD: INSTANTIATE ARCHITECTURAL ELEMENTS AND ALLOCATE RESPONSIBILITIES

4.5.1 Primary A and B

The primary and backup elements of both A and B have the same behavior. The behavior of A alone is described here.

- The element A receives messages from both query and update clients. It updates its state based on the update client messages and replies to queries from the query clients.
- Element A is normally deployed on the same processor as the backup copy B' of the element B. Just after a failure occurs to B, B' is promoted, and both A and B occupy the same processor until a new version of B is started. The process of switching the primary B to the just-started element B is not defined.
- Element A sends a heartbeat to the health monitor every 0.25 seconds.
- Element A copies its state to `CheckpointFileA` every minute.
- Element A accumulates the state changes made due to update client messages and writes them to `LogFileA` every 1.0 seconds. This write is synchronized with sending the checkpoint.
- The start-up of A and A' was not addressed, since there is another team tackling this issue.
- The proxy element will receive a request that both copies of the element A have failed, will stop sending updates, and will notify the necessary actors.

4.5.2 Persistent Storage

There are four persistent storage files: `CheckpointA`, `CheckpointB`, `LogFileA`, and `LogFileB`. All new values of these files overwrite the old values.

4.5.3 Health Monitor

The health monitor uses a timer to check whether it has received the heartbeat from A, B, A', and B'. If it fails to receive a heartbeat before the timer expires, it notifies the proxy.

4.5.4 Asynchronous Communication

The asynchronous communication mechanism receives a request from the update clients to a method (for example, A.a), and directs the request to the appropriate element.

1. The mechanism sends the name server the method A.a and receives the access code to the proxy element for A.a.
2. The mechanism sends the update message to the proxy element A.a.
3. When the mechanism receives the forward request for A.a to send the message to AA.a, it sends the request to AA.a and caches the handle for AA.a.
4. Any subsequent requests are made directly to the AA.a handle.
5. When a failure occurs, the mechanism receives the forward request to AA'.a and uses that handle for subsequent requests.

6. If AA.a fails and there is no standby, the mechanism informs the update client to stop sending updates.

4.5.5 Synchronous Communication

The synchronous communication element receives requests from the query clients and has almost the same behavior as the asynchronous communication element. The only difference is that it blocks the query client until it receives the answer to the query, which it then sends to the query client.

4.5.6 Proxy

The proxy element does most of the work in causing a smooth transition to the backup when the primary fails. It does the following:

1. The proxy service registers all the methods associated with both A and B with the naming service.
2. The proxy service starts AA, AA', BB, and BB' and registers all their methods with the naming service. It creates a cache by mapping the names used by the clients (e.g., A.a) and the names created by the elements (e.g., AA.a and AA'.a). It determines which element is primary and which is secondary.
3. The proxy service is called by either the synchronous or asynchronous communication element when a client requests a service; for example, A.a. It replies with a "forward request" to AA.a if AA is the primary.
4. When the health monitor signals the proxy that the primary (e.g., AA) has failed, it sends a forward request to both the synchronous and asynchronous communication elements to access all the standby methods (e.g., AA'.a), thus promoting AA' to be primary.

4.5.7 Update Clients

The failure of a primary component (e.g., A) and the switchover to A' are transparent to the update clients. Any updates sent during the window between failure and restoration are lost. But the timing window has been analyzed to be small enough for the Track Manager to continue working even with these lost messages. When the primary component fails and there is no backup, the update client will be notified and will stop sending updates until the service is restarted.

4.5.8 Query Client

The failure of the primary component when there is a backup is once again transparent to the update clients. They will have to wait slightly longer for an answer to their query, but that time has been evaluated as acceptable. When the primary component fails and there is no backup, the query client will be notified and will stop requesting queries.

The interfaces have been defined throughout Step 4 in Section 4 but are captured here for consistency and convenience. Note that some of the interfaces that were defined in the first iteration (see Section 3.2) are not repeated in Table 19.

Table 19: Summary of Interfaces

From Element	To Element	Interface	Timing Conditions	Descriptive Sections
Primary A	CheckpointA	Update state	60 seconds	4.5.1
Primary A	LogFileA	Log changes	1 second	4.5.1
Primary A	Health Monitor	Heartbeat	0.25 seconds	4.5.1
Primary B	CheckpointB	Update state	60 seconds	4.5.1
Primary B	LogFileB	Log changes	1 second	4.5.1
Primary B	Health Monitor	Heartbeat	-	4.5.1
CheckpointA	Primary A	Update state	During recovery	4.5.1
LogFileA	Primary A	Log changes	During recovery	4.5.1
CheckpointB	Primary B	Update state	During recovery	4.5.1
LogFileB	Primary B	Log changes	During recovery	4.5.1
Health Monitor	Proxy	Primary fail-ure	Within 1 second of detection	4.5.3
Query Client	Synchronous Communica-tion	Request for service	5 per second	4.5.8
Proxy	Naming	Registration of A, B, A', B' services	During start-up	4.5.6
Proxy	Synchronous Communica-tion	Primary failed (A or B)	During recovery	4.5.6
Proxy	Asynchronous Communica-tion	Primary failed (A or B)	During recovery	4.5.6

4.6 STEP 7 OF ADD: VERIFY AND REFINE REQUIREMENTS AND MAKE THEM CONSTRAINTS

In Step 7, we verify that the decomposition of the fault-tolerance services supporting the Track Manager element meets the functional requirements, quality attribute requirements, and design constraints, and we show how those requirements and constraints also constrain the instantiated elements.

The architectural drivers are shown once more in Table 20 and were used in one or more pattern selections (except for scenario 3). The restart scenario was not explicitly used, since the restart design was being done in parallel and a later merging was anticipated.

Table 20: Architectural Drivers

#	Architectural Drivers	Defined in Section	Applies to Pattern Choices
1	Scenario 1 Quick Recovery	2.3	Restart, Deployment, Data Integrity, Fault Detection,
2	Scenario 2 Slow Recovery	2.3	Data Integrity
~~3~~	~~Scenario 3~~ ~~Restart~~	~~2.3~~	~~Not used~~
4	Requirement 1 Track Manager Functionality	2.1	Restart
5	Requirement 2 Checkpoint to Persistent Storage	2.1	Deployment
6	Design Constraint 1 Spare Capacity	2.2	Fault Detection
7	Design Constraint 2 Two Replicas	2.2	Restart, Deployment
8	ADD Step 1, #1 Deployment Characteristics	3.2	Restart, Deployment, Data Integrity
9	ADD Step 1, #2 Communication Mechanisms	3.2	Update Client Behavior Query Client Behavior
10	ADD Step 1, #3 Checkpoint Timing	3.2	Data Integrity

Notes:

1. The breakdown of the timing requirements allocation derived from scenario 1 is shown in Table 18 on page 31.
2. The additional capabilities required by the elements defined prior to this step (Track Manager, query clients, update clients, persistent storage, synchronous communications, and asynchronous communications) are all defined in Section 4.5. The naming service and registration service required no extensions.
3. The responsibilities of the two new elements (proxy service) and (health monitor) are fully described in Section 4.5.

4.7 STEP 8 OF ADD: REPEAT STEPS 2 THROUGH 7 FOR THE NEXT ELEMENT OF THE SYSTEM YOU WISH TO DECOMPOSE

Now that we have completed Steps 1 through 7, we have a decomposition of the fault-tolerance service (specifically, the Track Manager system element). We generated a collection of responsibilities, each having an interface description, functional requirements, quality attribute requirements, and design constraints. You can return to the decomposition process in Step 2 where you select the next element to decompose. In our case, we do not have child elements to further decompose.

5 Summary of the Architecture

The architecture developed up to this point is described in this section. Also included are reminders about the parallel designs underway that this architecture must be resolved with and the issues that must still be tackled.

5.1 ARCHITECTURE SUMMARY

A summary of the design is given below.

- The Track Manager has two components (A and B), deployed on two hardware platforms. Each hardware platform contains the primary of one component and the secondary of the other component. The failure of A and B has equivalent activities, and the failure of Z=A or Z=B is discussed below.
- The primary and secondary components (A, A' and B, B') will give a heartbeat every 0.25 seconds to a health monitor.
- A persistent storage mechanism is required to save the state of the components every minute. Each component will write its state separately to persistent storage as `CheckpointA` and `CheckpointB`. These writes are synchronized to occur with the heartbeat.
- Each primary component will cumulate all its update changes for one second and then send the bundle to a `LogFile` in persistent storage. Each component will synchronize the sending of its `LogFile` with the heartbeat. There will be a `LogFileA` and a `LogFileB`.
- When the health monitor determines that a failure in Z has occurred, it informs the proxy service and the standby component Z' of the failure. The proxy service also sends a "forward request" to both the asynchronous and synchronous communication components indicating that Z' is the new primary.
- When the standby component Z' receives the signal that its primary has failed, it asynchronously reads the `CheckpointZ` and `LogFileZ` from persistent storage. It computes the new state of Z, which it saves to persistent storage. It is now ready to respond to client requests.
- When the synchronous communication component receives the "forward request" signal from the proxy, it sends its next update to the new primary. It does not know (or care) if previous updates were received.
- When the asynchronous communication component receives the signal from the proxy, one of the following situations occurs:
  - It has no active requests, so it sends the next update to the new primary.
  - It has an active update underway, so it sends the current request to the new primary.
- When a new standby replica is started, a primary is already in place.
- When a new replica is started and promoted to primary, it reads `CheckpointZ` and `LogFileZ` and updates the state of A. It then proceeds to handle update requests but ignores query requests. Many updates are indeterminate because of the downtime. Some help from the operator is needed to bring the system back to a reasonable state, at which time queries

will be resumed. This design has many overlaps with the initial start-up design, but the details of the start-up are not part of this design effort.

5.2 DESIGN ISSUES BEING RESOLVED ELSEWHERE

Some designs are being resolved elsewhere:

- the mechanisms for the persistent storage
- the start-up procedures for A and B and their coordination

5.3 REMAINING DESIGN ISSUES

This report provides an overview of how to make the Track Manager fault tolerant and to satisfy the architectural drivers, especially the end-to-end timing requirement. Some views of the software have been captured, but views are missing, such as class diagrams, layered views, container views, use cases, and many sequence diagrams detailing the information embedded in the various "Reasoning" and "Implications" sections of Section 4.4.3. In particular, a state transition diagram that shows the details of how the software elements respond together to provide the desired behavior is missing. This model is the only way to ensure that there are no "timing windows" in the behavior.

In addition, four design issues need to be addressed:

- How do the health monitor and proxy elements recover from failures in the software or hardware, and how they are distributed?
- The proxy service does not know which clients have a service request underway and which do not. How does a client react to a "forward request" when it has no request underway?
- How does the health monitor know about the proxy?
- How does the system respond to a failure in a secondary component?

6 Comments on the Method

While working with the example application described in this report, we made the following observations:

1. The person doing the design was familiar with the fault-tolerance concerns and alternative patterns used in the example. This designer was also familiar with the ways of reasoning about selecting between alternatives and the timing model needed to evaluate the effectiveness of the choices.
2. The documentation of the ADD results is awkward at times, since we documented development according to the structure of the ADD method. The result was rather clumsy documentation with Section 4 having four levels of indentation and the other sections having only one or two. However, in most real software architecture description documents, the documentation structure will not be dependent on the development method (ADD) but rather on the most effective way of capturing the views developed.
3. The developer chose to develop the architecture for fault tolerance in a single iteration, which resulted in too many alternative patterns to represent comfortably as a matrix. The pros and cons of each pattern were also not detailed explicitly, but were embedded within the rationale for making a selection within each pattern alternative.
4. ADD weakly implies that the development of an architecture is done sequentially—at each iteration, an element is chosen for design elaboration, all architectural drivers are known before starting the design of the element, and this iteration's results are then used in the next iteration. In development of large-scale architectures, this in unlikely to happen. Different architects (or architecture teams) will be assigned to different elements of the architecture and will work in parallel; that situation will require cooperation between the architects working on different elements and an eventual merging of the resulting designs.
5. The author placed sufficient information about the discriminating parameters in the results of the first iteration, especially Table 4 and Table 5. In practice, this is not usually the case; rather, the fault-tolerance expert would discover, during Step 4 when evaluating some pattern, that sufficient information was not available and would have to estimate the values as described in ADD Section 7.1, points 2a and 2b.

7 Conclusion

This report demonstrates how an experienced designer can use the ADD method to design fault tolerance into a system to satisfy some architectural drivers consisting of requirements, stakeholder-defined scenarios, and design constraints. The concerns, tactics, and patterns listed in the various tables in Section 4 are typical of those to be considered for such a design. In cases where the architectural drivers are more challenging, the design decisions change, but the basic approach can still be followed. Second and third iterations through the design cycle may be necessary. This report also clearly distinguishes between choosing well-known patterns during the first iteration and the alternatives for improvement during the second iteration. In the first iteration, straightforward choices were made from predefined patterns; whereas in the second, alternatives included changing a number of design aspects to achieve the improved qualities.

Glossary

architectural driver	An architectural driver is any functional requirement, design constraint, or quality attribute requirement that has a significant impact on the structure of an architecture.
architectural patterns	Architectural patterns are well-known ways to solve recurring design problems. For example, the Layers and Model-View-Controller patterns help to address design problems related to modifiability. Patterns are typically described in terms of their elements, the relationships between elements, and usage rules.
architectural tactics	Architectural tactics are design decisions that influence the quality attribute properties of a system. For example, a Ping-Echo tactic for fault detection may be employed during design to influence the availability properties of a system. The Hide Information tactic may be employed during design to influence the modifiability properties of a system.
candidate architectural driver	Candidate architectural drivers are any functional requirements, design constraints, or quality attribute requirements that have a potentially significant impact on the structure of an architecture. Further analysis of such requirements during design may reveal that they have no significant impact on the architecture.
design concept	A design concept is an overview of an architecture that describes the major types of elements that appear in the architecture and the types of relationships between them.
design concern	Design concerns are specific problem areas that must be addressed during design. For example, for a quality attribute requirement regarding availability, the major design concerns are fault prevention, fault detection, and fault recovery. For a quality attribute requirement regarding availability, the major design concerns are resource demand, resource management, and resource arbitration [Bass 2003].
design constraints	Design constraints are decisions about the design of a system that must be incorporated into any final design of the system. They represent a design decision with a predetermined outcome.

discriminating parameter Discriminating parameters are characteristics of patterns that you evaluate to determine whether those patterns help you achieve the quality attribute requirements of a system. For example, in any restart pattern (e.g., Warm Restart, Cold Restart), the amount of time it takes to do a restart is a discriminating parameter. For patterns used to achieve modifiability (e.g., layering), a discriminating parameter is the number of dependencies that exist between elements in the pattern.

functional requirements Functional requirements specify what functions a system must provide to meet stated and implied stakeholder needs when the software is used under specific conditions [ISO/IEC 2001].

interface The interface for an element refers to the services and properties required and provided by that element. Note that *interface* is not synonymous with *signature*. Interfaces describe the PROVIDES and REQUIRES assumptions that software elements make about one another. An interface specification for an element is a statement of an element's properties that the architect chooses to make known [Bass 2003].

patterns See *Architectural Patterns.*

property A property is additional information about a software element such as name, type, quality attribute characteristic, protocol, and so forth [Clements 2003].

quality attribute A quality attribute is a property of a work product or goods by which its quality will be judged by stakeholders. Quality attribute requirements such as those for performance, security, modifiability, reliability, and usability have a significant influence on the software architecture of a system [SEI 2007].

quality attribute requirements Quality attribute requirements are requirements that indicate the degrees to which a system must exhibit various properties.

relationship A relationship defines how two software elements are associated or interact with one another.

requirements Requirements are the functional requirements, design constraints, and quality attribute requirements a system must satisfy for a software system to meet mission/business goals and objectives.

responsibility A responsibility is functionality, data, or information that is provided by a software element.

role A role is a set of related responsibilities [Wirfs-Brock 2003].

software architecture	The software architecture of a program or computing system is the structure(s) of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003].
software element	A software element is a computational or developmental artifact that fulfills various roles and responsibilities, has defined properties, and relates to other software elements to compose the architecture of a system.
stakeholder	A stakeholder is someone who has a vested interest in an architecture [SEI 2007].
tactics	See *Architectural Tactics*.

Bibliography

URLs are valid as of the publication date of this document.

[Avizienis 2004]
Avizienis, Algirdas; Laprie, Jean-Claude; Randell, Brian; & Landwehr, Carl Source. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing 1*, 1 (January/March, 2004): 11-33.

[Bass 2003]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 2003.

[Clements 2003]
Clements, P.; et al. *Documenting Software Architectures Views and Beyond*. Reading, MA: Addison-Wesley, 2003.

[ISO/IEC 2001]
International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 9126-1:2001, Software Engineering–Product Quality–Part 1: Quality Model*. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=22749 (June 15, 2001).

[OMG 1999]
Object Management Group (OMG). *Fault Tolerant CORBA*. http://www.omg.org/docs/orbos/99-12-08.pdf (December 1999).

[SEI 2007]
Software Engineering Institute. *Software Architecture Glossary*. http://www.sei.cmu.edu/architecture/glossary.html (2007).

[Siewiorek 1992]
Siewiorek, Daniel & Swarz, Robert. *Reliable Computer Systems: Design and Evaluation*. Burlington, MA: Digital Press, 1992.

[Wirfs-Brock 2003]
Wirfs-Brock, Rebecca & McKean, Alan. *Object Design Roles, Responsibilities, and Collaborations.* Boston, MA: Addison-Wesley, 2003.

[Wojcik 2006]
Wojcik, R.; Bachmann, F.; Bass, L.; Clements, P.; Merson, P.; Nord, R.; & Wood, B. *Attribute-Driven Design (ADD), Version 2.0* (CMU/SEI-2006-TR-023). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006. http://www.sei.cmu.edu/publications/documents/06.reports/06tr023.html.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	February 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Practical Example of Applying Application-Driven Design (ADD), Version 2.0	FA8721-05-C-0003

6. AUTHOR(S)

William G. Wood

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TR-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2007-005

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report describes an example application of the Attribute-Driven Design (ADD) method developed by the Carnegie Mellon® Software Engineering Institute. The ADD method is an approach to defining a software architecture in which the design process is based on the quality attribute requirements the software must fulfill. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

The example in this report shows a practical application of the ADD method to a client-server system. In particular, this example focuses on selecting patterns to satisfy typical availability requirements for fault tolerance. The design concerns and patterns presented in this report—as well as the models used to determine whether the architecture satisfies the architectural drivers—can be applied in general to include fault tolerance in a system. Most of the reasoning used throughout the design process is pragmatic and models how an experienced architect works.

14. SUBJECT TERMS	15. NUMBER OF PAGES
attribute-driven design, ADD, architectural drivers, software architecture, architecturally significant requirements, decomposition	58

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL